FSAF: File System Aware Flash Translation Layer for NAND Flash Memories*

Sai Krishna Mylavarapu[1], Siddharth Choudhuri[2], Aviral Shrivastava[1], Jongeun Lee[1], Tony Givargis[2]

Sai.Mylavarapu@asu.edu

[1]Department of Computer Science and Engineering, Arizona State University, United States; [2]Center for Embedded Computer Systems, University of California, Irvine, United States

Abstract

NAND Flash Memories require Garbage Collection (GC) and Wear Leveling (WL) operations to be carried out by Flash Translation Layers (FTLs) that oversee flash management. Owing to expensive erasures and data copying, these two operations essentially determine application response times. Since file systems do not share any file deletion information with FTL, dead data is treated as valid by FTL, resulting in significant WL and GC overheads. In this work, we propose a novel method to dynamically interpret and treat dead data at the FTL level so as to reduce above overheads and improve application response times, without necessitating any changes to existing file systems. We demonstrate that our resource-efficient approach can improve application response times and memory write access times by 22% and reduce erasures by 21.6% on average.

1. Introduction

Flash memory is a non-volatile semiconductor memory that is becoming ubiquitous with attractive features like low power consumption, compactness and ruggedness. USB memory sticks, SD cards, Solid State Disks, MP3 players, Cell phones etc. are some of the well-known applications of the flash memory technology. NAND flash markets have seen substantial growth in recent years and analysts predict that the trend will continue [3].

However, flash comes with challenges to be addressed. One of the most important of them is “Erase-before-Rewrite” property: once a flash cell is programmed, a whole block of cells needs to be erased before it can be reprogrammed, and the erase operation is an extremely time consuming process. To hide this latency from the application, another block is used as a temporary write buffer to absorb rewrites. A rewrite to a full buffer triggers a fold operation as to consolidate valid data in both blocks into a new block, freeing up the two old blocks. Also, as invalid data accumulates and the free space in the whole device falls below a critical limit after continued writes, a cleanup operation called Garbage Collection (GC) needs to be performed to regenerate free space by reclaiming invalid data by performing a series of forced fold operations.

Another challenge of flash is limited life time: NAND flash can survive only a specific number of program/erase cycles, typically 100,000. Wear Leveling (WL) needs to be performed so as to make sure that all the flash blocks are evenly erased and the device can make use of all available program/erase cycles. This process involves frequent data shuffling between highly erased and least erased blocks. Thus, both GC and WL operations involve expensive erasures and data copying, and so are very time and energy intensive. Thus, flash device delays, and hence application response times are heavily influenced by the efficiency of the above two algorithms.

To understand the impact of above two operations on application response times, we ran a digital camera workload on a 64MB Lexar flash drive formatted as FAT32 [13] and fed resulting traces to Toshiba NAND flash [17] simulator to measure WL and GC overheads. During the scenario a few media files of sizes varying between 2KB and 32MB were created and deleted. Application response times at various instances are plotted in Fig.1. The peak delays at the right extreme of the figure correspond to instances where a GC is being carried out on dead data. As media files were not updated, only invalid data was because of deleted files. Since file systems only mark deleted files at the time of deletion, but not actually erase corresponding dead data in flash, FTL treats dead data as valid until specifically overwritten by new file data. When free space is below a critical level, this might result in a costly GC operation as above. Thus, various file systems are shown to be lengthy in response times in the presence of dead data [5]. Table 1 lists overheads associated with wear leveling alone on dead data, in terms of percentage increase in device delay, erasures, average memory write access time and folds. It has to be noted that, unlike GC overhead that occurs only at higher flash utilizations, WL overhead is always present. Thus, dead data contributes significantly to flash delays by affecting GC and WL operations.

Since the file system data structure residing on flash is known only to the file system, FTL can only interpret reads and writes to flash, but not file operations like file creation

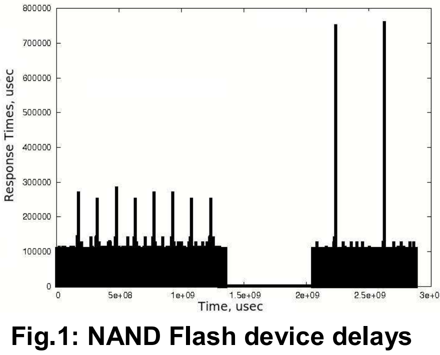

Fig.1: NAND Flash device delays

*This work was partially funded through grants from Consortium for Embedded Systems, Arizona State University.

978-3-9810801-5-5/DATE09 © 2009 EDAA

Table 1: Effect of dead data on various performance metrics

Metric	% increase due to dead data
Device Delays	12
Erasures	11
W-AMAT	12
Folds	14

and deletion, and hence unaware of dead data corresponding to deleted files. On the other hand, sharing file system data with FTL is not possible without changing existing file system implementations. Thus, previous efforts [1][2] [8] [9] [12] [18] [19] have only focused on improving GC and WL efficiency, but have not attempted to take dead data into consideration or necessitated [5] a change in existing system architecture. In this paper, we propose a comprehensive approach, FSAF: File System Aware FTL that enables FTL to recognize file deletion dynamically and resource-efficiently, without necessitating any changes to existing file systems. File deletions are interpreted by FSAF by observing changes to the file system data structure in flash. We also propose a mechanism to proactively handle dead data to significantly reduce GC and WL overheads. Experimental results demonstrate that FSAF can improve application response times and average write access times by 22% on an average, besides reducing erasures by 21.6% and significantly reducing the number of folds and GCs.

2. Background

Flash is organized into blocks and pages. A block is a collection of 32 pages each of 512 bytes. Each page has a 16 byte out-of-band (OOB) area used for storing metadata. In addition to read and write, flash also has erasure operation. Owing to the "Erase-before-rewrite" characteristic, a re-write to a page is possible only after the erasure of the complete block it belongs to. Available blocks in Flash are organized as Primary and Replacement blocks [5]. When a page rewrite request arrives, a primary block is assigned a replacement block. When the replacement block itself is full, and another rewrite is issued, a fold or merge operation needs to be performed: valid data in old two blocks is consolidated and written to a new primary block and the former are freed subsequently. Also, after a series of rewrites, free space in the device falls below a critical limit and needs to be regenerated by garbage collecting the invalid data. At the end of this GC process, valid data is consolidated into primary blocks. Thus, a GC is a series of forced fold operations. GC is a very time consuming operation, involving lengthy erasures and valid data copying, and may take as long as 40sec [10].

Another limitation of flash is endurance: it can only withstand finite number of erasures, typically 100,000. In other words, to address the endurance of flash, WL makes sure that all the blocks are erased uniformly, and avoiding localized wear. This operation involves identifying most worn out and least worn out blocks, and swap data between these on a periodic basis. Thus, WL also is a time consuming process, affecting application response times considerably. In order to unburden applications from overseeing various aspects of flash management as described above, a dedicated Flash Translation Layer (FTL) [7] may be employed, that enables existing file systems to use NAND flash without any modifications by hiding flash characteristics.

Fig. 2 depicts the file deletion operation in FAT32 file system. When a secondary storage like flash is formatted, FAT32 allocates first few sectors to FAT32 table to serve as pointers to actual data sectors. When a file is created or modified, the table is updated to keep track of allocated /freed sectors of the file. However, when a file is deleted or shrunk, the actual data is not erased. In over-writable media like hard disks, this poses no problem, as the new file data is simply overwritten over dead data. However, because flash doesn't allow in-place updates, dead data resides inside flash until a costly GC or fold operation is triggered to regain free space. Also, WL operation is carried out by FTL regularly on dead data blocks. Thus, dead data results in significant GC and WL overhead, affecting application response times.

3. Related Work

Several works to improve application response times have been proposed so far, by attempting to improve the efficiency of GC and WL operations. The greedy GC approach was investigated by Wu et al. [18]. Kawaguchi et al. [9] came up with the cost-benefit policy, by considering both utilization and age of blocks. Cost Age Time (CAT) policy [12] was considered by Chiang et al. that also focuses on reducing the wear on the device (increase endurance) apart from addressing segregation. Kim et al. [8] proposed a cleaning cost policy, which focuses on lowering costs and evenly utilizing flash blocks. A swap-aware GC policy [14] was introduced by Kwon et al. In order to minimize the GC time and extend the lifetime of the flash based swap system, they implemented a new Greedy-based policy by considering different swapped out time of the pages. Various approaches to improve WL efficiency have been proposed [2], where wear leveling is achieved by recycling blocks with small erase counts. Static wear leveling approaches were also pursued [19] to treat level both non-cold and cold data blocks.

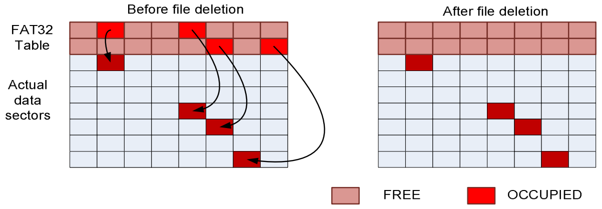

Fig.2: File deletion in FAT32 file system

Kim et. Al [5] proposed a new file system, MNFS, to address uniform write response times by carrying out block erasures immediately after file deletions. This method necessitates changes to existing system architectures.

We recognize that the key to improving application response times without necessitating any changes to existing file systems is to enable FTL to detect and treat dead data dynamically. To this end, in this work, we propose an FTL-based framework to efficiently recognize and also handle dead data. We have chosen to demonstrate our results on FAT32 file system, but the method is equally applicable to all other file systems that perform implicit file deletions.

4. FSAF: File System Aware FTL

FSAF monitors write requests to FAT32 table to interpret any deleted data dynamically, subsequently optimizing GC and WL algorithms accordingly. Also, depending upon the size of dead content and the flash utilization, proactive dead data reclamation is carried out.

4.1. Dead Data Detection

Dead data detection is carried out by FSAF dynamically as files are deleted by the application. Since the file system does not share any information with the FTL regarding file management, the only way we can interpret file system information at FTL is by understanding the formatting of flash and keep track of changes to the file system data structure residing on flash. The goal of dead data detection is to carry out this process efficiently without affecting performance.

The format of flash can be understood by reading the first sector on flash, called Master Boot Record (MBR) and the first sector in the file system called FAT32 Volume ID. The LBA_Begin field of the MBR reveals the location of the FAT32 Volume ID sector. Subsequently, the location of the FAT32 table can be determined as follows:

$$FAT32\_Begin\_Sector = LBA\_Begin + BPB\_RsvdSecCnt$$

The size of the FAT32 table is given by the field $BPB\_FATSz32$. Both $BPB\_RsvdSecCnt$ and $BPB\_FATSz32$ are read from the FAT32 Volume ID sector.

Once the size and location of the FAT32 table are determined, dead sectors can be recognized by monitoring writes to the table. FAT32 stores the pointer to each data sector allocated to a particular file in corresponding locations in the FAT32 table. To delete a particular file, all the pointers to data sectors are freed up by zeroing out their content. In other words, dead sectors resulting from shrinking or deleting a file can be found out by reading corresponding pointers prior to their zeroing out. If all the sectors in a block are dead, the whole block is marked as dead.

Thus, FSAF needs to maintain a buffer for reading FAT32 table sector before it is zeroed out by the file system. Fig. 3 depicts the algorithm.

4.2. Avoidance of Dead Data Migration

Once dead sectors are recognized, GC and WL algorithms are instructed to avoid copying their content during regular operation of flash. Thus, dead data migration is avoided during valid data copy occurring while carrying out GC and WL operations.

4.3. Proactive Reclamation

When larger files or files occupying contiguous sectors are deleted, dead data occupies complete blocks. Since these blocks do not contain any valid data, they can be reclaimed without any copying costs, unlike blocks that require valid data copy during normal folding operation. Thus, reclaiming such blocks is inherently a highly efficient operation in comparison to a forced fold operation during a GC. Thus, when the free space in flash falls below a critical threshold, instead of proceeding with costly GC operation, dead blocks can be reclaimed to delay or avoid GC by regenerating free space dynamically.

However, application response times still might suffer when all the dead data is reclaimed together, owing to costly erasure operations. In order to avoid this, proactive reclamation of dead blocks is taken up. FTL triggers GC higher flash utilizations [9], i.e., when the free space in the device is below a critical limit, and continues folding until free space reaches another threshold. In other words, to avoid delays due to GC, free space in the device should be kept above the GC threshold. So, dead block reclamation should be scheduled when flash utilization is reasonably

```
dead_data_detection():
    1.  Calculate size and location of FAT32 Table by
        reading MBR and FAT32 Volume ID sectors
    2.  Monitor writes to FAT32 Table
    3.  If a sector pointer is being zeroed out, mark
        corresponding sector as dead
    4.  Mark a block as dead if all the sectors in the
        block are dead
```

Fig.3: Dead data detection.

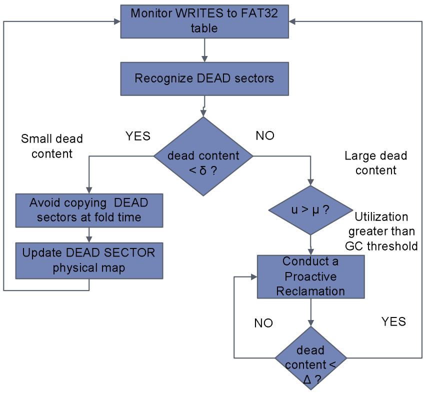
Monitor WRITES to FAT32 table
Recognize DEAD sectors
Small dead content
YES
dead content < δ ?
NO
Large dead content
Avoid copying DEAD sectors at fold time
Update DEAD SECTOR physical map
u > μ ?
Utilization greater than GC threshold
Conduct a Proactive Reclamation
NO
YES
dead content < Δ ?

Fig.4: Proactive reclamation.

high, but not high enough to trigger a GC operation. On the other hand, number of dead blocks proactively reclaimed must be as small as possible, as expensive erasure operations can impact application response times. Yet another important factor to be taken into consideration is the amount of dead data in flash - this decides whether or not proactive reclamations need to be run.

The proactive reclamation algorithm is as presented in Fig. 4. We first check whether the dead content is greater than a threshold δ. If not, GC and WL are informed to avoid useless dead data migration by marking dead sectors. If dead content is greater than δ, we check whether system utilization is higher than μ, i.e. whether at least μ percentage of blocks is already used. In such a case, we proceed to reclaim dead blocks proactively apart from avoiding dead data migration. Thus, dead block reclamation proceeds until number of dead blocks reach another threshold Δ.

Even though proactive reclamation improves application response times by avoiding or delaying costly GC operation, it should be scheduled in such a way that doing so itself does not penalize application a lot. Since proactive reclamation is a series of erase operations, it can be time consuming. In other words, parameters δ, μ and Δ should be carefully configured such that reclamation is highly efficient. Large values for δ and μ avoid frequent reclamation, but might impose a lot of reclamation activity. Small values for Δ mean smaller reclamation activity, but frequent triggers for reclamation. To arrive at reasonable values for these parameters, we explored the effect of varying these parameters on various performance metrics, as presented in the next section. The results confirm our intuition at best performance is achieved at high values of δ and μ low values of Δ.

5. Results and Discussion

5.1. Experimental Setup

We used trace-driven approach for the experimentation. A 64MB flash memory stick was formatted as FAT32 and three benchmarks representing various file system deletion activities were run on the same. In order to extract detailed traces of benchmarks, the USB stick was accessed through our FAT32 implementation. A trace of flash accesses, along with timing, access type and sector information along with the actual data being written was generated for each of the following benchmark:

s1: Huge sized file creation and deletion

s2: Medium sized file creation and deletion

s3: Small sized file creation and deletion

These benchmarks represent most frequently encountered scenarios on removable flash storage media such as SD cards in applications like digital cameras, mp3 players, digital camcorders and memory sticks. The collected traces were fed to a simulated Toshiba NAND Flash [17]. We realized log-based NFTL [8] on top of it and realized greedy [18] approach. FSAF was finally integrated with the setup.

In order to simulate real-world scenarios, we brought flash to 80% utilization and the size of flash for each benchmark was set to 64 MB. FTL was configured to start GC when the number of free blocks falls below 10% of total number of blocks and stop GC as soon as percent free blocks reaches 20% of total number of blocks. WL is triggered whenever the difference between maximum and minimum erase counts of blocks exceeds 15. The size of files used in various scenarios was varied between 32MB to 2KB.

5.2. Configuring FSAF Parameters

The parameters δ, μ and Δ need to be configured to run FSAF. We ran proactive reclamation algorithm with various values of δ and μ for all the benchmarks, and results supported our intuition that higher values for these parameters result in higher performance. By setting these to high as possible, proactive reclamation is triggered only when the system is low in free space, but runs frequently enough to generate sufficient free space. Thus, δ was set to 0.2 and μ to 0.85, i.e. when the dead data size exceeds 20% of the total space and system utilization is 85%, proactive reclamation is triggered.

To determine the best value for Δ, we observed variation in the total application response times, number of erasures, and GCs against various sizes of reclaimed dead data, represented by δ` ( = (δ – Δ)). Owing to lack of space, related results were omitted. We observed that when δ` was increased from 0 to 0.18, flash delays and erasures decrease initially and increase afterwards, as the reclamation activity increases. However, number of GCs remains the same. Thus, δ` needs to be set to a small positive value. This concurs with our hypothesis that small values for δ` are better than large values. So, Δ was set to 0.18.

In essence, FSAF is configured to proactively reclaim dead data as soon as dead content becomes more than 20% of the total flash size when flash utilization is greater than 85%, and reclaims 2% of dead blocks at each invocation.

5.3 Improvement in Application Response Times

Fig.5 depicts total application response times for each of the benchmark for both greedy and FSAF approaches. We observe that the FSAF approach improves response times by 22% on the average, and 32% for the scenario *s2* compared to the greedy approach. to greedy implementation. From Fig. 5, we can observe that there is a variation in the total response times for different scenarios, owing to the content and distribution We observe that maximum gains can be obtained when dead data occupies contiguous rather than randomly distributed sectors, as in the scenario *s2.* However, we see that FSAF achieves 22% improvement on the average.

It has to be noted that the total device delay includes delays incurred due to reads, writes issued by the application as well as those issued during carrying out GC and WL activity. When file system issues reads and writes and folding and wear leveling are triggered, additional writes and reads to pages and OOBs are issued by the FTL during the process of valid data copying. In other words, total writes carried out are more than application-issued writes. Since FSAF always avoids dead data migration and directly reclaims dead blocks, device delays are reduced, contributing to the reduction of flash access times and hence application response times.

Fig. 6 depicts average memory write access times (W-AMAT) for different scenarios for both greedy and FSAF approaches. We can observe that improvements in W-AMAT after employing FSAF are similar to improvements in response times. This is because of the fact that read access times of flash are much lower than write access times, and also because reads are normally cached. The variation in the average write access time across benchmarks is owing to dead data content.

It has to be noted that the response times suffer majorly at higher flash utilizations when GC operations are triggered out to regenerate free space. So, if enough free space can be generated at higher utilizations, we can delay or even avoid costly GCs. FSAF achieves the same by dead data reclamation at higher utilizations. On the other hand, WL overhead because of dead data, which is incurred at all flash utilizations is avoided by FSAF by avoiding dead data migration.

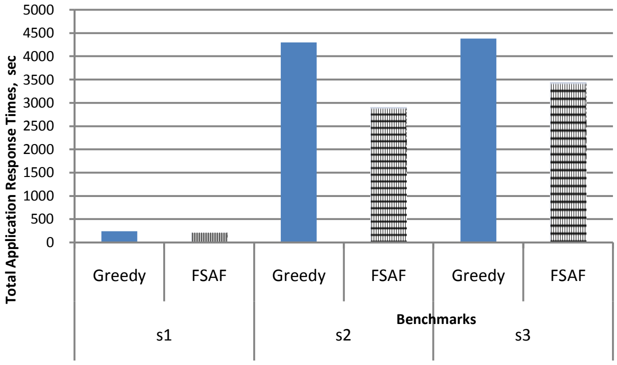

Fig.5: Total application response times for various benchmarks

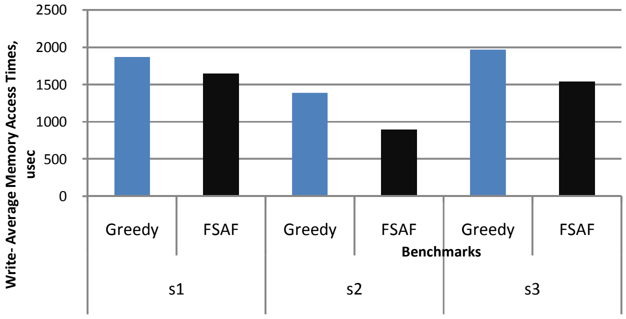

Fig.6: Average memory write-access times for various benchmarks

5.4 Improvement in GCs and Erasures

Table 2 provides improvements with respect to the number of GCs, erasures and folds for each benchmark, for both greedy and FSAF methods. The most important observation from this table is that, on an average, FSAF reduces number of erasures by 21.6%, by avoiding erasures associated with wear leveling dead data. Since erasures determine the life expectancy of flash, endurance is proportionally improved. Reduced number of erasures also means significant energy reduction, as an erasure is the costliest of all flash memory operations.

Also, GCs are also reduced by 43% on the average compared to greedy method. This is achieved by generating enough free space in the device by performing proactive reclamation. In other words, this means the elimination of undesirable peaks in the device response times depicted in Fig.1. Similarly, folds are reduced by employing FSAF. By reclaiming dead blocks proactively, FSAF eliminates the need for creating replacement blocks for dead blocks, and thus, unnecessary fold operations are eliminated.

It has to be noted that FSAF approach also results in lesser algorithmic overhead. An FTL triggers GC upon free block count reaching certain critical threshold. At such an instance, blocks are sorted by a metric decided by the GC policy, and are subsequently reclaimed in the sorted order until enough free blocks are generated. For example, in Greedy approach, blocks are sorted by their dead page count. FSAF, on the other hand directly erases dead blocks, i.e., blocks only with maximum benefits, doing away with costly sorting operation. The benefits associated manifest themselves in the reduction of the number of erasures, and improved GC efficiency by reducing number of writes and reads during folding.

It has to be noted that FSAF gains are heavily dependent upon the dead data content and distribution. However, since FSAF naturally switches to regular WL and GC operations when there is no dead data, its performance is at least as good as the normal case.

5.5 Overheads

The overhead associated with FSAF comes from dead data detection and proactive reclamation. To detect dead data, FSAF needs to monitor writes to only three sections of flash: the MBR, Volume ID and the FAT32 table itself. By reading and storing MBR and Volume ID at every format time, need for constructing formatting information at every flash plug-in is eliminated. To detect which sector is being deleted, FSAF needs to maintain a buffer of size of maximum one sector. Also, finding out which sector is

Table 2: Improvement in erasures, GCs and folds

	Erasures			GCs			Folds		
Benchmark	Greedy	FSAF	%Decrease	Greedy	FSAF	%Decrease	Greedy	FSAF	%Decrease
s1	4907	4347	**11.41**	10	7	**30.00**	2294	1979	**13.73**
s2	2631	1760	**33.11**	11	5	**54.55**	1249	792	**36.59**
s3	5384	4293	**20.26**	25	14	**44.00**	2541	1976	**22.24**

being deleted is an $O(s)$ operation, where s is the number of sector pointers stored in a single sector of the FAT32 table. Subsequent addition and deletion from the dead data list are all $O(1)$ operations. Thus, algorithmic overhead introduced by FSAF is only $O(s)$ per write. Since typically there are only 128 pointers per sector, this overhead is very minimal. Proactive reclamation, on the other hand, reduces the overall overhead on the system. Since proactive reclamation executes at a higher efficiency than a normal GC operation and also eliminates or delays regular GCs, effectively system overhead is significantly reduced.

6. Conclusion

In this paper, we proposed an FSAF: a file system aware FTL that can dynamically and efficiently detect dead content in flash. We showed that FSAF improves application response times significantly by treating dead data efficiently during GC and WL operations, and also performing proactive reclamation to delay or even avoid costly GC operations. The proposed approach results in significant overall improvement in flash management, by also decreasing number of erasures and write access times. The solution is realized without necessitating any file system changes and comes with a minimal resource overhead. Results obtained by running various benchmarks show that FSAF also improves longevity of flash by reducing the number of erasures significantly. As a further step, FSAF can be improved by scheduling proactive reclamation in the background when the application is idle.

7. References

[1] A. Ban. Flash file system. United States Patent, no.5404485, April 1995.

[2] A. Ban. Wear leveling of static areas in flash memory. US Patent 6,732,221. M-systems, May 2004.

[3] Elaine Potter, “NAND Flash End-Market Will More Than triple From 2004 to 2009”, http://www.instat.com/press.asp?ID=1292&sku=IN0502461SI

[4] Richard Golding, Peter Bosch, John Wilkes, “Idleness is not sloth”. USENIX Conf, Jan. 1995

[5] Hyojun Kim, Youjip Won , “MNFS: mobile multimedia file system for NAND flash based storage device”, Consumer Communications and Networking Conference, 2006. CCNC 2006. 3rd IEEE

[6] Hanjoon Kim, Sanggoo Lee, S. G., “A new flash memory management for flash storage system,” COMPSAC 1999.

[7] Intel Corporation. “Understanding the flash translation layer (ftl) specification”. http://developer.intel.com/.

[8] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho. “A space-efficient flash translation layer for compactflash systems”. IEEE Transactions on Consumer Electronics, May 2002.

[9] A. Kawaguchi, S. Nishioka, H. Motoda, “A Flash-memory Based File System”, USENIX 1995.

[10] Li-Pin Chang, Tei-Wei Kuo, Shi-Wu Lo, “Real-Time Garbage collection for Flash-Memory Storage Systems of Real-Time Embedded Systems”, ACM Transactions on Embedded Computing Systems, November 2004

[11] V. Malik, 2001a.” JFFS—A Practical Guide”, http://www.embeddedlinuxworks.com/articles/jffs guide.html.

[12] Mei-Ling Chiang, Paul C. H. Lee, Ruei-Chuan Chang, “Cleaning policies in mobile computers using flash memory,”
Journal of Systems and Software, Vol. 48, 1999.

[13] Microsoft, “Description of the FAT32 File System”, http://support.microsoft.com/kb/154997

[14] Ohoon Kwon, Kern Koh, “Swap-Aware Garbage collection for NAND Flash Memory Based Embedded Systems”, Proceedings of the 7th IEEE CIT2007.

[15] M. Rosenblum, J.K. Ousterhout, “The Design and Implementation of a Log-Structured FileSystem,” ACM Transactions on Computer Systems, Vol. 10, No. 1, 1992.

[16] S.W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S.W. Park, H.-J. Songe. “FAST: A log-buffer based ftl scheme with fully associative sector translation”. The UKC, August 2005.

[17] Toshiba 128 MBIT CMOS NAND EEPROM TC58DVM72A1FT00, http://www.toshiba.com, 2006.

[18] M. Wu, W. Zwaenepoel, “eNVy: A Non-Volatile, Main Memory Storage System”, ASPLOS 1994.

[19] Yuan-Hao Chang, Jen-Wei Hsieh, Tei-Wei Kuo, “Endurance Enhancement of Flash-Memory Storage, Systems: An Efficient Static Wear Leveling Design”, DAC’07